

[webmonkey](#)/programming/

PHP from the Ground Up

by [Tim Ziegler](#)

PHP is a powerful scripting language that fits gracefully into HTML and puts the tools for creating dynamic websites in the hands of the people — even people like me who were too lazy to learn Perl scripting and other complicated backend hoodoo.

This tutorial is for the person who understands HTML but doesn't know much about PHP. One of PHP's greatest attributes is that it's a freely distributed open-source language, so there's all kinds of excellent reference material about it out there, which means that once you understand the basics, it's easy to find the materials that you need to push your skills.

I discovered PHP when I was building a site called LostRock.com. Since I didn't have a budget to hire programmers to create a beefy backend, I decided to teach myself how to build it with PHP. As I stumbled along, I was struck by how intuitive and straightforward the language was — you'll see, just keep reading.

Before we begin, you will need to install a server on your own machine in order to test your PHP scripts locally. (If you have space on a Web server which supports PHP, you can also test your PHP there, but this is kind of a pain because it means you'll need to FTP your files or telnet in every time you want to change something. If you're not sure whether your site host supports PHP, just ask 'em.) To do so, simply follow Julie's article about [installing a Web server with PHP](#). After you've done that, come back and we'll dive in by step-by-stepping through the basics.

What Is It?

So, what is this whole PHP business all about?

PHP is a program that gets installed on top of your Web server software. It works with versions of [Apache](#), Microsoft IIS, Netscape Enterprise Server, and other server software packages.

You use PHP by inserting PHP code inside the HTML that makes up your website. When a client (anybody on the Web) visits a Web page that contains this code, your server executes it. That's why you need to install your own server in order to test PHP locally — the server is the brain here, not your browser. Users don't need any special plug-ins or anything to see your PHP in action — it gets to the end user as regular old-fashioned HTML. The experts tell me that PHP is nice because it does not put a big strain on your server's CPU. (I like it because it makes me feel smart.)

PHP is a scripting language, like HTML. That means that code does not need to be compiled before it gets used — it gets processed on the fly as necessary.

Before we dig in, you should know about a site called [PHP.net](#). PHP is an open-source language, and PHP.net is its control center, with extensive reference material about the language and tips sent in by users across the globe. PHP.net has exceptional, deep information about the language, but it can be a little cryptic for the newcomer. We'll look more closely at how to use PHP.net at the end of this

tutorial.

So, what kinds of things can PHP do? Welllll ... it can:

- take info from Web-based forms and use it in a million ways (store it in a database, create conditional pages depending on what the forms said, set cookies for later, send email, write your mom on her birthday);
- authenticate and track users;
- run threaded discussions on your site;
- serve different pages to people using different browsers or devices;
- publish an entire website using just a single layout template (server-side includes -style);
- serve XML pages.

Plus, it can display either the clown or the chicken man depending on which box you check:

Do you want to see the chickenman?

- Yes!
- No! Show me the clown.

Show Me!

But before we can get to the specific uses of PHP, we need to start with a quick preview of the building blocks of PHP, beginning with a sample script. This example script is titled "chickenman.php." When called by a Web browser, it would simply read, "I am the CHICKEN MAN!"

```
<?php  
  
print ("I am the CHICKEN MAN");  
  
?>
```

Everything in red is the PHP code, the black is a string of text that you're telling it to "print," as in "display on a Web page," not "print this on a printer."

The `<?php` and `?>` tags start and end a PHP script, and your meat goes in the middle. Got that? Good! Now let's walk through the basic rules you need to know to before you can write your first PHP script.

The Basics

The code itself fits right inside a page's HTML, and like HTML it is made up of plain ol' text. So a page that displays the words "I am the CHICKEN MAN!" message would sit inside an HTML page named *something.php*, like this:

```
1.  
2. <html>  
3. <head>  
4. <title> Chicken Man Example </title>  
5. </head>  
6.  
7. <body>  
8.  
9. <font color="red">My PHP code makes this page say:</font>  
10.  
11. <p>  
12.  
13. <?php  
14.
```

```
15. print ("I am the CHICKEN MAN");
16.
17. ?>
18.
19.
20. </body>
    </html>
```

See how that works? The HTML is rendered as regular HTML, but everything inside the `<?php` and `?>` tags gets processed as PHP.

Basic Syntax

It's time to write your own first PHP script. The basic rules of PHP are as follows:

Naming Files

In order to get a PHP script working, the file it's in or the file that it calls to do the heavy lifting must end in `.php` (earlier versions used the file extensions `.php3` and `.phtml`). Like HTML, your files are saved as plain text.

Comments

It's important to get in the habit of leaving notes about your code with the comment tags so that months down the road you can make sense of what you were trying to make your script do. The way you set comments apart from your code (that you don't want displayed or executed) is with either `///
"//" at the beginning of each line, or surrounded by /* and */ if you want to comment out several lines:`

```
<?php

// This will be ignored. Note to self:
// Pick up ice cream, cake, and balloons.

print ("I am the CHICKEN MAN");

/*
This, too, will be ignored.
Hey, and don't forget
the spanking machine!
*/

?>
```

Code Syntax

Start of Code

Every piece of PHP code begins with `<?php` (or the abbreviated `<?` if your server is configured to handle that).

End of Code

The way to signify that the PHP code is finished is by adding `?>` at the end.

Every Chunk

With a few exceptions, each separate instruction that you write will end with a semicolon.

Parentheses

The typical function looks like this ...

```
print ( );
```

... where "print" is the function and the stuff that the function works on sits inside the parentheses, with a semicolon to finish it off. (Just to confuse you, "print" is the exception that also works without parentheses.) By the way, `echo ()` is the same as `print ()`.

Much like HTML, the actual formatting of your PHP code (where you put spaces, line breaks, etc.) will not affect the outcome *except* those parts of the code that tell a Web browser how to display your page. So this piece of code ...

```
<?php  
  
print ("I am the CHICKEN MAN");  
  
?>
```

... is effectively identical to:

```
<?php print ("I am the CHICKEN MAN"); ?>
```

Like more complicated HTML, it behooves you to use white space and tabs in your code to make the code more understandable. (Behoove you too, pal!)

Capisce? Ready to write your first script? Let's go.

Your First Script

OK, so write your first script already! Copy the following script, but put whatever you want inside the quotation marks. "Print" here means print to the screen of a Web browser when you open the file:

```
<html>  
<body>  
  
  <?php  
  
    print ("I am the CHICKEN MAN");  
  
  ?>  
  
</body>  
</html>
```

Save the file with any name that has no spaces and ends in `.php`, and if you've installed a server on your own machine, you need to save the script somewhere inside the server's root folder (on Windows this is typically in the "wwwroot" directory inside the "inetpub" directory on your C: drive).

The next step is to open the file in your Web browser. Since you need the server to run your PHP code, you have to open the file through a URL that finds the correct file through your Web server. On Windows, you find this by looking at the Network settings in your Control Panel. Under the "Identification" tab you'll see "Computer name." Your computer name is your root URL. My computer name is "rocketboy," so to see the contents of my root directory, I type "http://rocketboy" into the Web browser and voila! I see the contents of my root folder. To open the file "chickenman.php" in a directory called "tests" inside the root directory, I'd type "http://rocketboy/tests/chickenman.php"

and see my example.

If you're testing on a PHP-able Web server on the Net, FTP your files anywhere on your server and they should work when you open them through the URL.

Go on now and get your first script working. Then come back and we'll have some fun. Together. (If you can't get your first script working, look at our [First Script Troubleshooting Guide](#).)

Error Messages

Fun, eh? Fun if it worked. If not — If you had an error in your script — you probably got an error message that looked something like this:

```
Parse error: parse error in C:\Inetpub\wwwroot\webmonkey_article\test9.php
on line 12
```

Error messages can be very useful and you're bound to run into lots of them. You'll get a message like this for every line in your script that has an error. For our purposes, all we really need to know is that there is something wrong with our code in line 12 of the document "test9.php," so let's look at that line and see if we can figure it out (good text editors like HomeSite have a function that lets you jump to any particular line). I always start by looking to see if my basic syntax is correct: did I leave out the closing tag, a line's semicolon, quotation marks?

Keep on Truckin'

Let's continue by adding to your test code from the last page to show a couple useful tools.

In the same code that you wrote before, drop in a couple more statements. As you see, you can gang up more than one PHP function inside the same opening and closing tags. My comments in the code explain what each part does:

```
<html>
<body>
```

```
This text right here (or any HTML I want to write) will show up just before
the PHP code stuff.
```

```
<p><p>
```

```
<?php
```

```
// first, this $PHP_SELF thang is
// an environment variable that'll show the
// path from your root folder to this
// document itself, like /webmonkey_article/test3.php.
// I put this in just for fun.
// NOTE: This may only work if your server is Apache.
```

```
print "$PHP_SELF";
```

```
// next we have to "print" any
// HTML code we want the browser
// to follow to determine
// the layout of the results page.
// In this case, we're adding a <p> tag
// the <p> tags could have been put
// inside the same print statement as the
// "I am the CHICKEN MAN" text.
```

```
// between the $PHP_SELF text and the
// next bunch of stuff.

print("<p>");

print("I am the CHICKEN MAN");

print("<p>");

/* This next "phpinfo" business outputs a long page that tells you exactly
how your version of PHP is configured. This can be useful when
troubleshooting problems down the road */

phpinfo();

?>

</body>
</html>
```

NOTE: Phpinfo will output a long page of info about your version of PHP. You don't need to understand what it all means, I just wanted to show you that it's there if you ever need it.

Very Able Variables

So far, all we've done is have a PHP script print some text. Big whoop. Let's get down and dirty now with variables, which are like the wind beneath the wings of the soaring eagle that, um, is PHP.

A variable is a container for holding one or more values. It is the means by which PHP stores information and passes it along between documents and functions and such. You may remember variables from algebra — in the equation $x + 2 = 8$, x is a variable with the value 6.

The reason why variables are so important to PHP is that the very notion of having dynamic Web pages — pages which respond somehow to user input — relies on data being passed around between pages (or parts of a page). Variables are the main mechanism for transferring data like this.

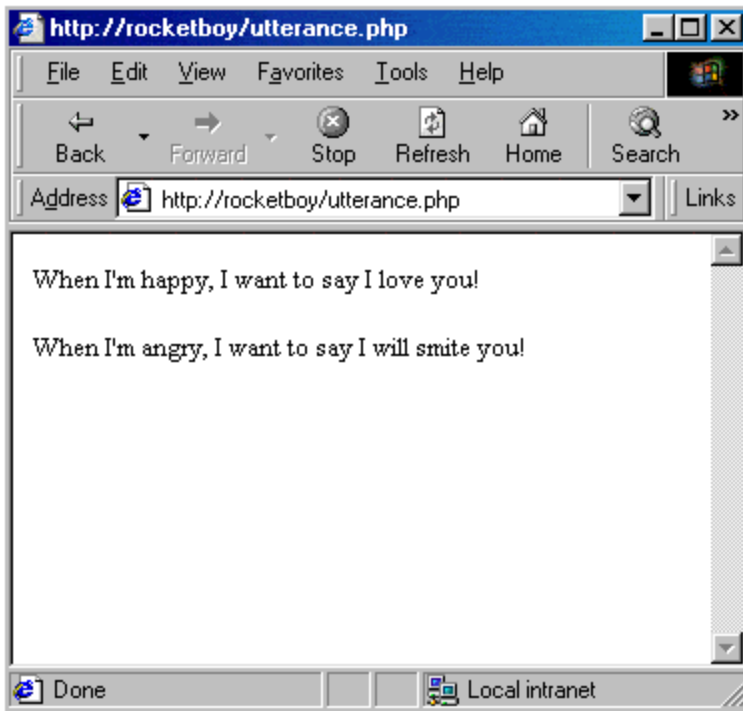
I think the easiest way to explain how variables work in PHP is to show them in action. There are three basic things you can do with variables:

1. Set them (give them one or more values);
2. Re-set them if they were set before;
3. Access them (read the value of a variable and then do something useful with it).

Variables in PHP start with a dollar sign ("\$"). Below I am setting a variable, using it, then setting and using it again. The value that a variable holds can be changed any time at all.

```
1. <?php
2. $utterance = "I love you!";
3. print ("When I'm happy, I want to say $utterance");
4. print ("<p>");
5. $utterance = "I will smite you!";
6. print ("When I'm angry, I want to say $utterance");
7. ?>
```

Here's what that page will look like:



In line two I have created a variable that I decided to name "utterance." All variables start with "\$", so my variable is written "\$utterance" in the code. Here's how that last code snippet breaks down line by line.

- Line 1 tells the browser: "Start PHP code here".
- Line 2 creates the variable \$utterance and also sets it, giving it the initial value of "I love you!".
- Line 3 prints a phrase that draws on the variable \$utterance.
- Line 4 creates a <p> tag in HTML to put vertical space between the two utterances.
- Line 5 RE-SETS the variable \$utterance and gives it the value "I will smite you!".
- Line 6 prints a new phrase that draws on the new meaning of the variable \$utterance.
- Line 7 tells Mr. Browser: PHP code ends here.

See how the variable \$utterance is used as a sort of container that can hold different values? We just set and then called variables inside the same script, but the power of PHP is that you can set a variable in one place — say from a form that a user fills out — and then use that variable later.

The syntax of setting a variable is to:

- define it with the = sign (\$utterance = "I love you!");
- use quotation marks if you're defining it with a string of letters ("I love you!"; numbers don't require quotes);
- end each instruction with a semicolon.

Then you call it by referring to the variable name (\$utterance in lines 3 and 6 — notice no quotation marks there).

That word utterance is starting to sound awfully funny isn't it? Utterance utterance utterance utterance utterance!

Naming Variables

You can name a variable anything you want so long as it follows these rules:

- it starts with a letter;
- it is made up of letters, numbers, and the underscore character (that's the `_` character, as in `"$baby_names"`);
- it isn't used elsewhere (like `"print"`).

Warning: Variable names *are* case-sensitive, so `$baby_names` and `$Baby_names` are not the same. You also should try to make your names have some meaning so that you can still make sense of your code next year.

In the examples so far, we have set variables as chunks of text, which are known as "strings." Variables can also hold the values of numbers and some other things as well (objects, arrays, booleans).

Final note: One thing that can be a little confusing when starting to use PHP is the use of quotation marks inside PHP functions. Use single or double quotes to set off strings (that is, chunks of text), as in:

```
print ("I am the CHICKEN MAN");
```

This will print the text I am the CHICKEN MAN. If you want to display quotation marks as characters in the output of your PHP script, you must escape them with the `"\"` character, which tells PHP not to use the next character as part of the code. So to output the text "I am the CHICKEN MAN" (with quotation marks showing in the result) the code would look like:

```
print (" \"I am the CHICKEN MAN\" " );
```

HTML Forms and PHP

In our examples so far, we have set variables and then used them all in the same code. This doesn't make much sense because in those instances, we could have just hard-coded the values instead of using variables.

Let's get some real mileage by creating HTML forms to gather user input, turning that input into variables, and then doing various things with the information that we just collected. Maybe I'm a dork, but I think this is fun.

No sense in sitting around waiting — let's go ahead and make a Web page that collects your favorite dirty word and displays it on another page that tells you what a pervert you are. All of this gives a page that looks a lot like [this](#).

First, we make the form page, which is regular HTML with a form in it. I'm calling mine "badwords_form.html," but call yours whatever you like. (If you want a good primer on HTML forms, read Jay's [Good Forms](#) tutorial.)

```
1. <html>
2. <head>
3. <title>My Form</title>
4. </head>
5. <body>
6.
7. <form action="bad_words.php" method=post>
8.
9. My name is:
10. <br> <input type="text" name="YourName">
11.
12. <p> My favorite dirty word is:
13. <br> <input type="text" name="FavoriteWord">
```

```

14. <p>
15.
16. <input type="submit" name="submit" value="Enter My Data!">
17. </form>
18.
19. </body>
20. </html>

```

This is a regular HTML form. The important pieces are as follows:

Line 7: the HTML that reads `action="bad_words.php"` tells the browser which PHP document will process the results of the form. That is to say, in a minute you'll create a document called "bad_words.php" which will be the little engine that makes the result page happen. (We'll get to the `method=post` part later on.)

Line 10: `input type="text"` determines that the form element which we want here is "text" or a text box (we could also have a radio button, check box, etc.); `name="YourName"` determines that whatever the user types into the text box will become a variable that we have called "YourName." This is what ties together forms and variables — each form field can set a variable to be used however you want.

Line 13: here you have another text input that sets a variable called "FavoriteWord" which is the user's favorite dirty word.

Line 16, 17: This code makes a submit button with the text "That's Right!" and ends the form.

Okeedoke. So this form will collect the unassuming user's name and favorite bad word, but now what do we do with it? Let's take the variables she set and echo them back in another context on another page.

On line 7 of the HTML above, we told the form to head on over to `bad_words.php` once the submit button was hit. This is what `bad_words.php` looks like:

```

1. <html>
2. <head>
3. <title>Perv!</title>
4.
5. </head>
6.
7. <body bgcolor="#FFFFFF" text="#000000">
8.
9. <p>
10.Hi <?php print $YourName; ?>
11.
12.<p>
13.You like the word <b> <?php print $FavoriteWord; ?> !?! </b>
14.
15.<p>You oughta be ashamed of yourself!
16.
17.</body>
18.</html>

```

See how this form passed a variable along from the form page to the PHP file? You have not seen the last of *this*.

Get versus Post

So far, we've used the "Post" method of passing form data as opposed to the other method, "Get."

This is the part of the form code that reads `<form action="bad_words.php" method=post>`.

The difference between these two is that the "post" method transparently passes along all the information the page has gathered, whereas the "get" method will pass all that info along as part of the URL (in the form above, this would look like:

`http://rocketboy/webmonkey_article/bad_words.php?YourName=bob&`

`FavoriteWord=crikey%21&submit=Enter+My+Data%21` — see how the info the user entered about his name and his favorite word get added to the URL?)

Arrays of Light

One of your best tools now that you've mastered variables — you have, haven't you? — are arrays.

Arrays give you the ability to store not just one value inside a variable, but a whole bunch of values in a single variable. Imagine! Why on earth would you want to do that? We'll get to that.

If I wanted to catalog all of the animals in my house, I could set each one as a regular variable. I've got two dogs, Phoebe and Ruby, and a squirrel that died in the attic last year, whom we'll call Rotty (the smell was no picnic, let me tell you). Setting each one as a variable looks like this:

```
$dog1 = "Phoebe";
$dog2 = "Ruby";
$squirrel1 = "Rotty";
```

But an array will let us store all these inside one single variable, which I'll call `$critters`. Each element of the variable has its own "key" that is used to access that part of the array, which can either be a string of letters or numbers.

Let me explain the "key" concept another way: If we're storing three different values inside one variable (like storing Phoebe, Ruby, and Rotty inside `$critters`), we need some way to be able to suck out any individual part of the array to use it. An array will automatically number each element that comprises it, so the key can be element 1, element 2, and element 3. Or, as we'll see later on, we can name each part of the array with text. In this case I could make the keys "fat dog," "skinny dog," and "squirrel" and use those to identify each array member.

Let's make a simple array and then use it. The easiest way to create an array is to use the `array()` function, which assigns a bunch of values to your array at once and looks like this:

```
$critters = array ( "Phoebe", "Ruby", "Rotty" );
```

This stores all my animal names into one variable (`$critters`) in an array, and automatically assigns a numbered "key" to each element starting in order and giving the first element the number 0. So Phoebe is element [0], Ruby is [1], Rotty is [2], etc. I make up the name of the array myself (here it's `$critters`).

You can now get at any of the array elements by referring to the variable followed by the element number in square brackets: `$critters[0]`, for example. Here it is in action:

```
<?php
print "$critters[2]";
?>
```

This will simply print the third element in the array, which is `Rotty` (don't forget that array numbers start at 0, so `$critters[2]` is third after `$critters[0]` and `$critters[1]`).

There's another way to set an array, or even to add to an existing array, by setting each element individually:

```
$critters[] = "Phoebe";
$critters[] = "Ruby";
$critters[] = "Rotty";
```

This'll have the same effect as using the `array()` function, giving the first element the key `[0]` and so on. But wait! I forgot about Opie the cat. Hmm. Regardless of how we made the array in the first place, I can easily add Opie like this:

```
$critters[] = "Opie";
```

PHP is smart enough to count the number of elements and give Opie the next available one, which in this case (after Phoebe, Ruby, and Rotty) is `[3]`.

To recap this concept, I can set an array to include the animals in my house either this way:

```
$critters[] = "Phoebe";
$critters[] = "Ruby";
$critters[] = "Rotty";
$critters[] = "Opie";
```

Or this way:

```
$critters = array ( "Phoebe", "Ruby", "Rotty", "Opie" );
```

Both will be indexed in the computer brain with the values:

```
$critters[0] = "Phoebe";
$critters[1] = "Ruby";
$critters[2] = "Rotty";
$critters[3] = "Opie";
```

And in both cases, you could get at any element in the array by describing its number ...

```
<?php
print "$critters[3]";

?>
```

... which would print the string `Opie` to the window of your browser.

Arrays can be made to do all kinds of things, like being incremented by number, sorted in alphabetical order, printed by different types of categorization, and many more.

Associative Arrays

Ready to get more complicated? The associative array indexes the contained elements not by numbers, but by names that you determine. Inside the `array()` function, you set up pairs where you name the key and its value using the combo of the "=" and the ">", like: `key=>"value"`. Here's what it looks like in action:

```
$PhoebeDog = array (
    name=>"Phoebe",
```

```

        description=>"fat dog",
        color=>"grey and white",
        age=>7
    );

```

Here we're telling the array to create the keys "name," "description," "color," and "age"; and we give each of those keys a value (name is "Phoebe", description is "fat dog," and so on).

We can get at any part of the array through the "key" names that we set, for example:

```
print $PhoebeDog[color];
```

will give us `grey and white`. We can also set each key individually, like so:

```

$animals[name] = "Phoebe";
$animals[description] = "fat dog";
$animals[color] = "grey and white";
$animals[age] = 7;

```

Finally, let's make it hurt. We're going to get some serious power out of this arrays business by creating a "multi-dimensional" array. A multi-dimensional array is an array (say the animals in my house) that is made up of other arrays (for each animal, an array that contains the critter's name, color, description, and age).

We make multi-dimensional arrays by creating one array:

```

$animals = array
(
    );

```

...and then we fill that array with an array of animals in which we've defined the keys, like this:

```

$animals = array (
    array ( name=>"Phoebe",
           type=>"dog",
           color=>"grey and white",
           age=>7 ),
    array ( name=>"Ruby",
           type=>"dog",
           color=>"brown and white",
           age=>7 ),
    array ( name=>"Rotty",
           type=>"squirrel",
           color=>"grey",
           age=>2 ),
    array ( name=>"Opie",
           type=>"cat",
           color=>"grey tabby",
           age=>5 )
    );

```

To use this now, we can get any part of the information contained in there by naming the overall array (`$animals`), naming the number of the sub-array that we want to find out about (Phoebe is `[0]`, Ruby is `[1]`, etc.) and then naming the key for the attribute we want to get at (name, type, color, or age).

To find out the age of the cat, we'd write:

```
print $animals[3][age];
```

Here's what it all looks like together. This is all in one page, but remember that you can set arrays in one place (in code or in form fields from another page or in a database, say) and get at the info contained within from somewhere else. Here I'm putting it all together on one page so you can see it all at once.

```
<html>
<head>
<title>Pet Arrays</title>
</head>
<body>

<?php

$animals = array (
    array ( "name" => "Phoebe",
           "type" => "dog",
           "color" => "grey and white",
           "age" => 7 ),
    array ( "name" => "Ruby",
           "type" => "dog",
           "color" => "brown and white",
           "age" =>7 ),
    array ( "name" => "Rotty",
           "type" => "squirrel",
           "color" => "grey",
           "age" =>2 ),
    array ( "name" => "Opie",
           "type" => "cat",
           "color" => "grey tabby",
           "age" => 5 )
    );

print $animals[2]["type"];
print ("<br>");
print $animals[3]["color"];

?>

</body></html>
```

Now [see what it does](#).

What we've just done is create an array that includes a sub-array for each animal which contains detailed info about that critter; then we print a sentence that uses the type and ages of two of the animals.

Operators; If, Else, Elseif; Loops

The whole deal about making dynamic websites is that you want your Web page to be as smart as possible — to have code sitting there that can make all sorts of decisions based on different kinds of user input, user conditions (what browser is Visitor X using?), or information that you set yourself. Examples of this could be:

- After a user enters an email address, check that it has a valid form (whoever@wherever.com)

and if not, serve a page that says, "hey pal, how about a VALID email address?"

- Serve one set of HTML to .com domains and another to .gov domains (where you try not to use any big words that might confuse 'em).
- Know whether a customer at your online store is meeting a minimum dollar amount for an online credit card purchase.
- And on and on and on — the possibilities are limitless.

If

The way to make your pages smart is to use *If*, *Else*, and *Elseif* statements along with *Comparison* and *Logical Operators*. The most important one of these is the `if` statement, which gives you the ability to code:

```
If some condition is true, then do somesuch thing;
If the condition is not true, then ignore it;
```

The syntax of this statement is as follows:

```
if (condition) {

    // code in here to execute if the condition is true

}
```

Here it is in action. First we set the variable `$FavoriteColor` to blue (line 3). Then we say "if `FavoriteColor` is blue, then print 'I like blue too!'"

```
<?php

$FavoriteColor = "blue";

if ($FavoriteColor == "blue") {

    print ("I like blue too!");

}

?>
```

Else

Else builds on the *if* statement as *if* to say:

```
If some condition is true, then do somesuch thing;
ELSE, in case that first condition is NOT true, then do this other thing.
```

It works like this:

```
if (condition) {

    // code in here to execute if the condition is true

} else {
```

```
// code in here to execute if the condition is not true  
  
}
```

Here it is in motion:

```
<?php  
  
$FavoriteColor = "yellow";  
  
if ($FavoriteColor == "blue") {  
    print ("I like blue too!");  
} else {  
    print ("You don't like blue?! Crazy fool!");  
}  
  
?>
```

What you see above is the typical format for writing statements like this. The key part is to see where the curly braces are so you don't get confused as to which statement belongs to which piece of the code. Above, the first set of { and } belong to the "if," the second { and } belong to the "else."

Elseif

There's one more sibling in the `if`, `else` family, and it's called `elseif`. Where `else` is sort of a blanket control that make something happen as long as the `if` statement is not true, `elseif` makes something happen if a specific condition is met:

```
IF some condition is true, then do somesuch thing;  
ELSEIF some other specific condition is true, then do another thing;
```

It looks like this:

```
<?php  
  
$FavoriteColor = "green";  
  
if ($FavoriteColor == "blue") {  
    print ("I like blue too!");  
} elseif ($FavoriteColor = green) {  
    print ("MMMMMMMMMMMMMMMM, green!");  
}  
  
?>
```

You could even add an ELSE statement at the end in case FavoriteColor were neither blue nor green.

Comparison and Logical Operators

We've seen how the "=" sign works when we set variables in the scripts we've written so far (as in the code `$FavoriteColor = "blue";` and `$utterance = "I will smite you!";`). The equal sign here is what we call the "assignment operator" and is the simplest operator we've got: `$a = b` means "the variable a is assigned the value b (for the moment)."

But you can squeeze a lot more juice out of your PHP using what are called "comparison operators," "logical operators," and "arithmetic operators." Here's what they are, in tables copied verbatim from [PHP.net](http://php.net).

Comparison Operators

These give you the ability to compare whether elements are equal, identical, less than or greater than one another (with some other variations).

Example	Name	Result
<code>\$a == \$b</code>	Equal	TRUE if \$a is equal to \$b.
<code>\$a === \$b</code>	Identical	TRUE if \$a is equal to \$b, and they are of the same type. (PHP 4 only)
<code>\$a != \$b</code>	Not equal	TRUE if \$a is not equal to \$b.
<code>\$a <> \$b</code>	Not equal	TRUE if \$a is not equal to \$b.
<code>\$a !== \$b</code>	Not identical	TRUE if \$a is not equal to \$b, or they are not of the same type. (PHP 4 only)
<code>\$a < \$b</code>	Less than	TRUE if \$a is strictly less than \$b.
<code>\$a > \$b</code>	Greater than	TRUE if \$a is strictly greater than \$b.
<code>\$a <= \$b</code>	Less than or equal to	TRUE if \$a is less than or equal to \$b.
<code>\$a >= \$b</code>	Greater than or equal to	TRUE if \$a is greater than or equal to \$b.

Logical Operators

Here you can compare elements using comparisons and, or, and the like.

Example	Name	Result
<code>\$a and \$b</code>	And	TRUE if both \$a and \$b are TRUE .
<code>\$a or \$b</code>	Or	TRUE if either \$a or \$b is TRUE .
<code>\$a xor \$b</code>	Xor	TRUE if either \$a or \$b is TRUE , but not both.
<code>! \$a</code>	Not	TRUE if \$a is not TRUE .
<code>\$a && \$b</code>	And	TRUE if both \$a and \$b are TRUE .
<code>\$a \$b</code>	Or	TRUE if either \$a or \$b is TRUE .

Aritmetic Operators

Just what it says — this is basic math.

Example	Name	Result

<code>\$a + \$b</code>	Addition	Sum of \$a and \$b.
<code>\$a - \$b</code>	Subtraction	Difference of \$a and \$b.
<code>\$a * \$b</code>	Multiplication	Product of \$a and \$b.
<code>\$a / \$b</code>	Division	Quotient of \$a and \$b.
<code>\$a % \$b</code>	Modulus	Remainder of \$a divided by \$b.

Loop-dee-Loops

Loops are really handy. They let you program your code to repeat itself (and also to figure out how many times to run through itself before it's done). Loops have instructions to "keep running this piece of code over and over again until certain conditions are met." You can use more than one kind of loop. Let's look at the most basic kind, the "while" loop.

The while loop says,

```
while (something is true)
{
    // do something that you specify
}
```

While loops are often used with incrementing and decrementing a variable that is an integer. What in the *hell* does that mean? It means that you can automatically have the script add (or subtract) a whole number (1, 2, 3, etc.) from part of a script each time it runs through, until the number reaches a maximum or minimum value that you've set.

So if you wanted a script to print the numbers from 1 to 10, you can tell it (this is English, not PHP here):

- a. the variable `$MyNumber = 1;`
- b. `print $MyNumber;`
- c. `add 1 to $MyNumber;`
- d. go to sep a. and run this script again with the new value of `$MyNumber;`
- d. stop when `$MyNumber` reaches 11;

The syntax for incrementing and decrementing the value of a variable is:

<code>\$a++;</code>	adds 1 to the value of the variable \$a each time through
<code>\$a--;</code>	subtracts 1 from the value of the variable \$a each time through

So the code itself for printing the numbers 1 through 10 could look like this:

```
1.
2. <?php
3.
```

```
4.  $MyNumber = 1;
5.
6.  while ($MyNumber <= 10)
7.
8.  {
9.
10. print ("MyNumber");
11.
12. $MyNumber++;
13.
14. }
15. ?>
```

line 1: start PHP code;

line 3: set variable \$MyNumber to 1;

line 5: initiate "while" loop: while the variable \$MyNumber is less than or equal to 10, execute what's below; otherwise move on;

line 9: print the current value of \$MyNumber;

line 11: add 1 to the value of \$MyNumber;

line 15: end PHP code.

To see what all this does, check out the [results of the code above](#).

For these loops, you can use all the "operators" listed on [page 9](#) to be the conditions that must be met before the loop stops.

Other Loops

PHP has other kinds of loops, but they are beyond the scope of this tutorial. If you want to learn what they are and how to use them, follow these links to PHP.net:

- [do..while loop](#)
- [for loop](#)
- [foreach loop](#)

Grand Master Functions

How are ya doing? Getting sleepy? We're almost done here, and if you've followed along this far you're doing great. Have some more coffee or whatever it is that keeps you perky, and we'll look at functions and then a few final thoughts.

If you've used HTML a lot, you know that it's a pretty limited language that was designed back in the dark ages before we all knew what the Net was capable of.

PHP, however, is very flexible. Not only does PHP have a library of canned functions that'll do everything from sorting stuff in alphabetical order to sending email, from connecting with databases to balancing your spaceship's inertial sub-space dampers, but you can also create your very own functions to do all manner of things related to your website. The functions you create get executed exactly like those from the PHP library, but they're your own. In the following section, I'll show a glimpse of how you create your own functions and feel the power.

Functions you create are like little machines that do something for you. You construct them and then call them as needed.

You'll remember that the very first thing we learned to do was a simple "print" statement, which followed the form:

```
<?php

print ("whatever it is I want to show up on-screen");

?>
```

Functions that you create are built upon a similar form, but take it farther:

```
<?php

function MyFunction ()

{

statements that make up the function;

}

?>
```

So you start a function with the words `function WhatYouNameIt()`, with the words `WhatYouNameIt()` being anything you choose (no spaces).

Then you define the rules of the function inside the following curly brackets (that's `{` and `}` on lines 5 and 9). Don't you just love the words "curly brackets?"

Let's walk through the making of a couple functions. Functions come in two flavors, those that require "arguments" and those that don't. An "argument" is a variable that comes from outside the function, but which the function needs in order to run.

Let's first look at one that doesn't require arguments:

```
<?php

function ChickenMan()

{

print "<b>I am the CHICKEN MAN!</b>";

}

ChickenMan();

?>
```

line 1: start PHP;

line 3: create function `ChickenMan`;

line 5: start definition of function `ChickenMan`;

line 7: definition of `ChickenMan` is to print "I am the CHICKEN MAN!" inside `` and `` tags;

line 9: end definition of `ChickenMan`;

line 11: call function `ChickenMan` (meaning "do the thing that we defined the function to do");

line 13: close PHP;

Any place in this Web page that you put "`ChickenMan();`" inside the `<?php` and `?>` tags, it'll print your little statement. See how this saves you time if you want to print this string a bunch of times in

different places?

Now let's get a little more complicated and create a function that does take arguments. Say I'm a loan shark and I'm going to loan you the money to buy that [personal hovercraft](#) you've been lusting after. I'll lend you however much you need, but you have to pay me 14 percent interest per week or I break your legs.

I'll create a function called `InterestPayment` which calculates your weekly payments (until you pay off the whole loan amount in one lump sum). If you're interested, take a look at [this script in action](#).

First we'll create a form page where the user enters how much of a loan he or she wants. The user enters the hovercraft's sticker price, and that number gets passed along from the form as a variable called `$Cost`. (For the HTML behind this, go [here](#).)

Next, our function will take the amount that the user enters into the variable `$Cost` and spit back 14 percent of that amount, which is how much the hapless borrower owes every week. This will happen in the PHP page titled "loanshark.php" that the form page points to (it points with this code: `<form action="loanshark.php" method=post>`). Here's what the function will look like:

```
1. <html>
2. <head>
3. <title>Loans</title>
4. </head>
5.
6. <body>
7.
8. <?php
9.
10. $interest_rate = .14;
11.
12. function YouOweMe($cost, $interest_rate) {
13.
14. $weekly_payment = ($cost*$interest_rate);
15.
16. print "You better pay me \$$weekly_payment every week, or else!";
17.
18. }
19.
20. YouOweMe($cost, $interest_rate);
21.
22. ?>
23.
24.
25. </body>
26. </html>
```

Here's the line-by-line breakdown.

line 8: start php;

line 10: set variable called `$interest_rate` to 14%;

line 12: create function `YouOweMe` that relates to the variables `$cost` and `$interest_rate`;

line 14: create variable `$weekly_payment`, the value of which is the cost times the interest rate;

line 16: print to screen a sentence that uses the value of the `$weekly_payment` variable;

line 20: engage the function `YouOweMe`, which (because of the function defined beforehand) simply prints the sentence `You better pay me [14% of the amount the user entered on the form page] every week, or else!`

Notice on line 16 that we want the code to actually print out a dollar sign before the weekly payment.

We escape that character with a `\` to make sure the PHP engine doesn't think we're naming a variable there. Then we just happen to name a variable right afterwards (`$WeeklyPayment`), so the two together look like `\$$WeeklyPayment`.

Some Final Thoughts

If it got a little brutal back there trying to make sense of my functions, don't worry — you'll probably get used to squinting through other people's code once you get going. That's because somebody has probably written code for whatever you need to do already, and is probably happy to share it with you. Making sense of it all gets easier and easier the more you use this stuff. And since PHP is an open-source language (meaning the source code is shared openly instead of being hoarded by an evil corporation), the language is surrounded by a sense of cooperation among its users, many of whom will help you if they can.

With the knowledge that you have so far, you can go almost anywhere with this PHP business. But how do you find out out to do a particular task that you need to do? Here are some good resources. These are just places that I frequent, but by no means is this a comprehensive list:

PHP.net

As I've mentioned, PHP.net is the language's central brain. Here are parts of PHP.net that I've found especially useful.

[Manual](#): The site contains a general manual for PHP. Some functions may take a while to figure out, but it's all here. Once at PHP.net, follow the links to Documentation --> View Online and pick your language.

Function List: At the top of the main php.net page you'll find a search box in which you can search different areas of the site. If there's a function you need to find, search through the functions there. Often you'll have to search for a word ("cookie") and then figure out from the results which function is what you want ("setcookie").

[FAQ](#): Find great info in the FAQ, including a category titled "Common Problems."

[Webmonkey's PHP collection](#)

[PHP Knowledge Base](#)

[PHPBuilder](#)

Recommended Reading

I bought several books when I started learning PHP that were of varying quality. I really liked Larry Ullman's "PHP for the World Wide Web" (Peachpit Press). "Teach Yourself PHP4 in 24 Hours" by Matt Zandstra was also good. Books published by O'Reilly are also generally excellent, though I haven't seen theirs on PHP.

How to Write Your Code

No matter what source you use for reference material (and you'll need sources!), there's a general rule of thumb to use when writing this stuff. It's always good to map out what you need to do and then test your concept as simply as possible. Once you get the machinery working, you can put in your real content and make it more and more complex. It's much harder to troubleshoot if you don't start out with simple pieces.

Enough of my blabbing. I hope you found this tutorial useful. Please feel free to email me if any parts were unclear or if you want to name your children after me. And just remember, the key ingredient to building any great website is love! Toodles!

Tim Ziegler is a Webmonkey contributing editor. He also builds websites in Texas and is the founder of FamilyAlbum.com

Copyright © 1994-2004 Wired Digital Inc., a Lycos Network site. All rights reserved.